Exact Analysis of the Cache Behavior of Nested Loops*

Siddhartha Chatterjee†
sc@cs.unc.edu

Erin Parker†
parker@cs.unc.edu

Philip J. Hanlon‡
hanlon@math.lsa.umich.edu

Alvin R. Lebeck§
alvy@cs.duke.edu

†Department of Computer Science
The University of North Carolina
Chapel Hill, NC 27599-3175

‡Department of Mathematics
University of Michigan
Ann Arbor, MI 48109

§Department of Computer Science
Duke University
Durham, NC 27708

ABSTRACT

We develop from first principles an exact model of the behavior of loop nests executing in a memory hierarchy, by using a nontraditional classification of misses that has the key property of composability. We use Presburger formulas to express various kinds of misses as well as the state of the cache at the end of the loop nest. We use existing tools to simplify these formulas and to count cache misses. The model is powerful enough to handle imperfect loop nests and various flavors of non-linear array layouts based on bit interleaving of array indices. We also indicate how to handle modest levels of associativity, and how to perform limited symbolic analysis of cache behavior. The complexity of the formulas relates to the static structure of the loop nest rather than to its dynamic trip count, allowing our model to gain efficiency in counting cache misses by exploiting repetitive patterns of cache behavior. Validation against cache simulation confirms the exactness of our formulation. Our method can serve as the basis for a static performance predictor to guide program and data transformations to improve performance.

1. INTRODUCTION

The growing gap between processor cycle time and main memory access time makes efficient use of the memory hierarchy ever more important for performance-oriented programs. Many computations running on modern machines are often limited by the response of the memory system rather than by the speed of the processor. Caches are an architectural mechanism designed to bridge this speed gap, by satisfying the majority of memory accesses with low latency and at close to processor speed. However, programs must exhibit good locality of reference in their memory access sequences in order to realize the performance benefit of caches.

Optimizing compilers attempt to speed up programs by performing semantics-preserving code transformations. Loop transformations such as *iteration space tiling* [62] are a major source of performance benefits. They restructure loop iterations in ways that make the memory reference sequence more cache-friendly. The theory of loop transformations is well-developed in terms of deciding the legality of a proposed transformation and generating code for the transformed loop. However, models of the expected performance gains of performing a given loop transformation are less well-developed [19, 38, 45, 48, 50, 51, 61]. Where such models exist, they are often heuristic or approximate. For example, tiling requires the choice of tile sizes, and the performance of a loop nest is typically a non-smooth function of the extents of the loop bounds, the tile sizes, and the cache parameters [13, 19, 38]. The model we develop in this paper can be used to quantitatively determine the number of cache misses of a proposed transformation without explicit simulation. Ultimately, such a model could be used to guide the choice of parameters in such program transformations.

A complementary method for improving sequential program performance that has been investigated in recent years is that of transforming the memory layout of its data structures. Such data layout transformations can vary in complexity; examples include transposition and stride reordering [32], array merging [39], intra- and inter-array padding [50, 51], data copying [38], and non-linear array layouts [14]. Once again, proper choice of parameter values is of paramount importance in getting good performance out of such transformations, but the models guiding this optimization are often inexact. For instance, Rivera and Tseng [50, 51] use heuristics to determine inter-array pad. However, there is empirical evidence that almost every choice of pad can be catastrophically bad for a program as simple as matrix transposition [16]. Better models are clearly needed to guide such optimizations. Our work in this paper is a step in this direction.

An aggressive form of data optimization is the use of certain families of *non-linear array layouts* that are based on interleaving the bits in the binary expansion of the row and column indices of arrays. Previous studies have demonstrated performance gains as well as robustness of performance resulting from the use of such layouts [14, 15]. Yet it is difficult to ascertain, short of simulation, the memory behavior of a program given a particular data layout. This paper works towards building an analytical model of cache behavior for such layouts that can provide insight into the relationship between such data layouts and memory behavior.

Our model is an alternative to the well-known Cache Miss Equations (CME) model of Ghosh *et al.* [26]. Compared to CME, our model has the following strengths and weaknesses.

*This work was supported in part by DARPA Grant DABT63-98-1-0001, NSF Grants EIA-97-26370 and CDA-95-12356, NSF Career Award MIP-97-02547, The University of North Carolina at Chapel Hill, Duke University, and an equipment donation through Intel Corporation's Technology for Education 2000 Program. Erin Parker is supported by a Lawrence Livermore Computer Science Graduate Fellowship. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

PLDI 2001 Snowbird, UT USA

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **2001**	2. REPORT TYPE	3. DATES COVERED **-**

4. TITLE AND SUBTITLE **Exact Analysis of the Cache Behavior of Nested Loops**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Defense Advanced Research projects Agency,3701 North Fairfax Drive,Arlington,VA,22203-1714**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**12**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

- Our model is *exact* as a consequence of our use of Presburger arithmetic as the underlying formalism. Ghosh *et al.* [26] use the abstraction of *reuse vectors* to simplify the analysis. Reuse vectors do not exist for all loop nests, and certainly do not exist in the presence of non-linear array layouts.

- Our model accurately determines *the state of the cache* at the end of executing a loop nest. This functionality is important for accurately counting compulsory misses [30], for rapidly leap-frogging up to a certain point in the computation, and for handling multiple loop nests.

- Our model handles *imperfect loop nests* in addition to perfect loop nests. We apply a transformation of Ahmed *et al.* [2, 3] to an imperfect loop nest, thereby converting it to a perfect loop nest with guards on statements. Ghosh *et al.* [26] consider only a single perfect loop nest.

- Our model handles a variety of *array layout functions*, from row- and column-major to non-linear. We will subsequently refer to row- and column-major layouts as *canonical* layouts [17]. The formulation for non-linear layouts is new, to the best of our knowledge.

- Our model handles caches with *modest levels of associativity* in a natural way. While Ghosh *et al.* [25] can handle set-associative caches, their solution method is equivalent to simulation in the worst case.

- Our model is capable of *symbolic analysis*. This is a direct consequence of our use of the Presburger formalism. For example, we can simplify a formula for the cross-interference between two arrays while keeping the difference of their starting addresses symbolic. The simplified formula can be rapidly evaluated for specific values of this variable.

- The enhanced capabilities of our model come at the cost of *computational complexity*, in the form of super-exponential worst case behavior of algorithms for satisfiability checking and quantifier elimination of Presburger formulas [60]. While we have a prototype implementation of our model as a SUIF [55] pass, and the analysis and formula generation portions of the implementation are acceptably efficient, significant improvements are necessary to the robustness and efficiency of the simplification and counting parts.

Compared to explicit simulation, our formulas capture temporal patterns of cache behavior that may not be apparent in simulation. Moreover, an analytical cache model provides deeper insight into the behavior than what may be learned from simulation. We anticipate that such information will make it possible to guide the choice of data layouts that optimize cache behavior. We validate the results of all our formulas against simulation in Section 4, thereby confirming their exactness.

The remainder of this paper is structured as follows. Section 2 reviews background material for discussing our approach to the cache analysis problem: basics of cache memory (Section 2.1), a new classification of cache misses (Section 2.2), the polyhedral model (Section 2.3), and Presburger arithmetic (Section 2.4). Section 3 constructs our model. Section 4 provides some preliminary results obtained using our cache analysis model. Section 5 discusses related work. Section 6 presents conclusions and future work.

2. BACKGROUND

This section provides background material and defines notation for the remainder of the paper.

2.1 Basics of memory hierarchies

We assume a simplified memory hierarchy that processes one memory access at a time, with no distinction between memory reads and writes.

2.1.1 Cache structure

The structure of a single level of a memory hierarchy—a *cache*—is generally characterized by three parameters [30]: **A**ssociativity, **B**lock size, and **C**apacity. Capacity and block size are in units of the minimum memory access size (usually one byte). A cache can hold a maximum of C bytes. However, due to physical constraints, the cache is divided into *cache frames* of size B that contain B contiguous bytes of memory—called a *memory block*. The associativity A specifies the number of different frames in which a memory block can reside. If a block can reside in any frame (*i.e.*, $A = \frac{C}{B}$), the cache is said to be *fully associative*; if $A = 1$, the cache is *direct-mapped*; otherwise, the cache is A*-way set associative*. A *cache set* is the group of frames in which a memory block can reside, and the *number of cache sets*, S, is given by $S = \frac{C}{AB}$.

We assume a two-level memory hierarchy, consisting of an A-way set associative cache with block size of B bytes and total capacity of C bytes followed by main memory. We also assume that main memory is large enough to hold all the data referenced by the program. The function $\mathcal{B}$ converts a memory byte address into a memory block address (with $\mathcal{B}(a) = \lfloor a/B \rfloor$). The function $\mathcal{S}$ converts a memory block address to the cache set to which it maps (thus, $\mathcal{S}(b) = b \bmod S$).

2.1.2 Cache dynamics

For an access to memory address m, the cache controller determines whether memory block $\mathcal{B}(m)$ is resident in any of the A cache frames in cache set $\mathcal{S}(\mathcal{B}(m))$. If the memory block is resident, a *cache hit* is said to occur, and the cache satisfies the access after its *access latency*. If the memory block is not resident, a *cache miss* is said to occur.

The *state of the cache* represents the memory block(s) contained in each set of the cache at any point during a program's execution. Thus, in a direct-mapped cache where each set holds one frame, the cache state $\mathbb{C}$ maps set s to the address of the memory block contained there. In general, $\mathbb{C}$ is a map from cache sets to the sets of memory blocks that they contain. $\mathbb{C}(s)$ is empty for a cache set s to which no block has been mapped.

2.2 Classification of cache misses

From an architectural standpoint, cache misses fall into one of three classes: *compulsory*, *capacity*, and *conflict* [30]. Capacity and conflict misses are often combined and called *replacement* misses. This classification is extremely useful for understanding the role of capacity and associativity in the performance of a cache; however, it does not have the property of *composability*.

Consider two program fragments P_1 and P_2, where, for $i \in \{1, 2\}$, fragment P_i incurs C_i cold misses and R_i replacement misses. Now consider the program fragment $P_{12} \stackrel{\text{def}}{=} P_1; P_2$ formed by sequential composition of P_1 and P_2, and suppose that it incurs C_{12} cold misses and R_{12} replacement misses. There is no simple relation connecting the misses of the whole to the misses of the parts. In particular, $C_{12} + R_{12} \neq C_1 + R_1 + C_2 + R_2$. Composition is a fundamental operation in the construction of programs and in the definition of programming language semantics. As we wish to count cache misses for individual program fragments and their compositions, we propose a different classification that is composable.

We classify misses from a program fragment into the following two classes.

- *Interior misses* are those data references that are guaranteed to miss, independent of the initial cache state when the fragment begins execution. In other words, given the code, the array layouts, and the structural parameters of the cache, such misses can be identified/enumerated/counted by analyzing the fragment in isolation.

- *Potential boundary misses* are those data references that may either hit or miss, depending on the initial cache state when the fragment begins execution. The potential occurrence of such misses can be identified by analyzing the fragment in isolation, but the actual occurrence of the miss can be determined only after considering the initial cache state.

Another equivalent view of this classification is that we can statically examine a program fragment in isolation and place each data memory access that it makes into one of three categories: those that are guaranteed to hit, those that are guaranteed to miss (interior misses), and those that could hit or miss depending on the initial cache state (potential boundary misses). In a second step, we further partition the potential boundary misses into hits and misses by resolving them against the cache state when the program fragment starts executing. We call these misses *boundary misses*. It follows that, in order to compose program fragments, we also need to determine the state of the cache after executing a program fragment. For a given program fragment P and an initial cache state S, we will let $\Psi(P, S)$ denote the final cache state after fragment P has completed execution.

Theorem 2.1 *Let program fragment P_1 executing from initial cache state $\mathbb{C}_0$ incur I_1 interior misses and $B_1(\mathbb{C}_0)$ boundary misses and produce final cache state $\mathbb{C}_1 = \Psi(P_1, \mathbb{C}_0)$. Let program fragment P_2 executing from initial cache state $\mathbb{C}_1$ incur I_2 interior misses and $B_2(\mathbb{C}_1)$ boundary misses and produce final cache state $\mathbb{C}_2 = \Psi(P_2, \mathbb{C}_1)$. Let program fragment $P_{12} \stackrel{\text{def}}{=} P_1; P_2$ executing from initial cache state $\mathbb{C}_0$ incur I_{12} interior misses and $B_{12}(\mathbb{C}_0)$ boundary misses and produce final cache state $\mathbb{C}_{12} = \Psi(P_{12}, \mathbb{C}_0)$. Then the following relations hold.*

$$\begin{aligned} I_{12} + B_{12}(\mathbb{C}_0) &= I_1 + B_1(\mathbb{C}_0) + I_2 + B_2(\mathbb{C}_1) \\ \mathbb{C}_{12} &= \mathbb{C}_2 \end{aligned}$$

PROOF. The proof follows immediately from the semantics of program composition and from the deterministic nature of the program fragments and of the cache. $\square$

Theorem 2.1 has several important consequences.

- The theorem enables the analysis of cache misses of a composite program fragment in terms of the cache miss behavior of its parts. Each part can be analyzed in isolation, and the results of these analyses can be combined using cache states. We will show later how to efficiently propagate cache state across a program fragment.

- Stronger assertions, like $I_{12} = I_1 + I_2$, do not hold in general.

- The theorem is silent about the nature of program fragments P_1 and P_2 or about how to calculate boundary and interior misses for them. In the remainder of the paper, we will choose loop nests as our atomic program fragments and use Presburger formulas to codify the various kinds of misses.

- The theorem provides additional leverage if symbolic analysis of the atomic program fragments is possible. For example, block-recursive codes [4] employ multiple dynamic instances of the same loop nest differing only in the starting addresses of the data arrays on which they operate. Symbolic analysis of such fragments would allow the cost of analysis to be amortized over multiple uses of the resulting formulas.

- Note that boundary misses for a fragment are bounded from above by the cache footprint of the data structures it accesses, which is in turn bounded from above by the number of cache frames. This number is typically much smaller than the number of interior misses. We could therefore avoid the calculation of cache state and approximate the number of cache misses of the composite program by $I_1 + I_2$, with an accompanying error bound.

2.3 The polyhedral model

Our model for analyzing cache behavior of loop nests is based on the well-known polyhedral model [20]. The program fragment whose cache behavior we are trying to analyze is a nested normalized loop with d levels of nesting, numbered 0 through $d-1$ from outermost to innermost. We first consider perfect loop nests; we will extend the model to imperfect loop nests in Section 3.4. The upper bound U_j of ι_j, the loop control variable (LCV) for loop j, is an affine function of the LCVs ι_0 through ι_{j-1}. The iteration space $\mathcal{I}$ is the set of all valid combinations of LCV values that are within the bounds of the loop nest. The notation $\ell = [\ell_0, \ldots, \ell_{d-1}]^T$ denotes a generic point in the iteration space $\mathcal{I}$. The iteration space possesses a total order $\prec$, which in the polyhedral model is the lexicographic ordering. The order specifies the temporal order in which the iteration points in the iteration space are executed.

The loop accesses elements of arrays $Y^{(0)}$ through $Y^{(m-1)}$. Array variable $Y^{(i)}$ has d_i dimensions, with n_j being the extent of the array in the $(j+1)^{\text{th}}$ dimension. The data index space $\mathcal{D}_i$ corresponding to array $Y^{(i)}$ is the Cartesian product $[0, n_0 - 1] \times \cdots \times [0, n_{d_i - 1} - 1]$.

The statements in the loop body make k references to array variables. The i^{th} reference R_i has three components: N_i, the name of the array referenced (so that $N_i = Y^{(j)}$ for some $j \in [0, m-1]$); F_i, the index expression of the reference, which identifies the coordinates of the array element accessed by this reference at iteration point ℓ; and S_h, the statement that contains reference R_i. To include statement S_h in the definition of reference R_i may seem excessive at this point, but it will be useful in Section 3.4 when we consider imperfect loop nests. The index expression F_i is constrained to be an affine function of ℓ in each of its components. Thus, F_i is a function from the iteration space $\mathcal{I}$ to the data index space $\mathcal{D}_{N_i}$.

Borrowing terminology from Ghosh *et al.* [26], we call a static instance of a memory read or write a *reference*, and a dynamic instance of that read or write an *access*. A reference and an iteration point uniquely define an access. The total order $\prec$ on iterations almost induces a similar total order on accesses; however, two accesses in the same iteration need to be ordered as well. We compose the total order $\prec$ on the iteration space and the order among references of an iteration to define a total order "precedes" (written $\lhd$) among accesses. Thus, access (R_i, u) precedes access (R_j, v) iff $(u \prec v) \vee (u = v \wedge i < j)$.

Several quantities are associated with array $Y^{(i)}$: a layout function $\mathcal{L}_i$, which is a 1-1 map from $\mathcal{D}_i$ into the memory address space $\mathbb{Z}_0^+$; μ_i, the starting byte address of the array; and β_i, the number of bytes per array element. Applying $\mathcal{L}_i$ to an element of the array

Object	Mathematical Representation
An iteration point	ℓ
ith array reference	$R_i = (Y^{(j)}, F_i, S_h)$
Access made by R_i at ℓ	(R_i, ℓ)
Array element accessed by R_i at ℓ	$e_i = Y^{(j)}[F_i(\ell)]$
Byte address of e_i	$m_i = \mu_j + \mathcal{L}_j(F_i(\ell)) \cdot \beta_j$
Block address of m_i	$b_i = \mathcal{B}(m_i)$
Cache set to which b_i maps	$s_i = \mathcal{S}(b_i)$

Table 1: Table of notation.

produces an offset, and multiplying the offset by β_i gives the byte offset from the starting address of the array in memory. Adding this offset to μ_i then gives the byte address of the element.

Putting all of this notation together, we have the objects of interest and their mathematical representations shown in Table 1.

Example 1 Consider the following loop nest for matrix multiplication, which we present in a stylized pseudo-code in an attempt to remain language-neutral.

```
do i = 0, n-1
  do j = 0, n-1
    do k = 0, n-1
S0:   C[i,j] = A[i,k]*B[k,j]+C[i,j]
    end
  end
end
```

This loop nest has depth $d = 3$. The LCVs are $\iota_0 = i$, $\iota_1 = j$, and $\iota_2 = k$. The loop nest accesses three arrays: $Y^{(0)} = A$, $Y^{(1)} = B$, and $Y^{(2)} = C$. Each array is two-dimensional, so that $\mathcal{D}_0 = \mathcal{D}_1 = \mathcal{D}_2 = [0, n-1] \times [0, n-1]$. There are four array references: $R_0 = A[i,k]$, $R_1 = B[k,j]$, $R_2 = C[i,j]$ (the read access), and $R_3 = C[i,j]$ (the write access). The index expressions of the four references are $F_0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, $F_1 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$, and $F_2 = F_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$. All references are contained in statement S_0.

2.4 Presburger arithmetic

Presburger arithmetic [31] is the subset of first order logic corresponding to the theory of integers with addition. Presburger formulas consist of affine constraints on integer variables, which can be either constraints of equality or inequality. The constraints are linked by the logical operators $\neg$, $\wedge$ and $\vee$, and the quantifiers $\forall$ and $\exists$. It has been used to model various aspects of programming languages, as well as in other areas such as timing verification [6, 7]. We use Presburger formulas to define polytopes whose contents describe interesting events like cache misses.

Presburger arithmetic is decidable; however, a quantifier elimination decision procedure has a superexponential upper bound on performance. More precisely, the truth of a sentence of length n can be determined within $2^{2^{2^{pn}}}$ time, for some constant $p > 1$ [46]. The bound is tight [60]. Bounded quantifier elimination has worst-case upper and lower bounds of $\Theta(2^{2^n})$ [60]. The complexity is related to the number of alternating blocks of $\forall$ and $\exists$ quantifiers [52] as well as to the numerical values of the integer constants and their co-primality relationships.

We use the Omega library [34] to manipulate and simplify our Presburger formulas, and have found its methods reasonably efficient for our applications.

3. THE CACHE ANALYSIS MODEL

The problem of central interest to us is the following.

> *Given a cache configuration as in Section 2.1, a loop nest $\mathbb{L}$ meeting the conditions of Section 2.3, the layout functions of the arrays accessed in $\mathbb{L}$, and an initial cache state $\mathbb{C}_{in}$:*
>
> - *count the interior misses incurred by $\mathbb{L}$;*
> - *count the boundary misses incurred by $\mathbb{L}$;*
> - *find the cache state $\mathbb{C}_{out}$ after execution of $\mathbb{L}$.*

A simple strategy to accomplish all of these goals is through simulation of the code. This is precisely what cache simulators [29, 40, 54, 56] do. The main drawback of simulation is its slowness: it takes time proportional to the running time of the code, usually with a significant multiplicative factor ($10 - 100$ is typical). In the matrix multiplication kernel of Example 1, this time is $\Theta(n^3)$. Our goal is to develop much faster algorithms, whose existence is suggested by the regularity of the array access patterns and the limited number of cache sets to which they map.

Section 3.1 provides the basic Presburger formulas necessary to describe the cache events in Section 3.2. Section 3.3 discusses how we count cache misses, given such Presburger formulas. Section 3.4 extends our model to analyze imperfect loop nests. Section 3.5 shows how to extend our formula for interior misses to handle modest levels of associativity. Section 3.6 reviews array layouts based on bit interleaving, and provides the Presburger formulas to describe them. Section 3.7 discusses issues related to physically indexed caches.

3.1 Describing cache structure using Presburger formulas

We now present the basic formulas that will be combined in Section 3.2 to describe cache events. The translations are mostly straightforward or well-known [18, 49].

3.1.1 *Valid iteration point*

The predicate $\ell \in \mathcal{I}$ describes the fact that iteration point $\ell = [\ell_0, \ldots, \ell_{d-1}]^T$ belongs to the iteration space.

$$\ell \in \mathcal{I} \stackrel{\text{def}}{=} \bigwedge_{i=0}^{d-1} 0 \leqslant \ell_i < U_i \tag{1}$$

3.1.2 *Lexicographical ordering of accesses*

When considering all accesses that occur before access (R_v, m), we include any access occurring at an iteration ℓ, such that $\ell \prec m$. To be complete, we must also include any access made at iteration m by a reference that occurs before R_v. The predicate $(R_u, \ell) \lhd (R_v, m)$ describes the fact that the memory access made by reference R_u at iteration ℓ precedes the memory access made by R_v at m.

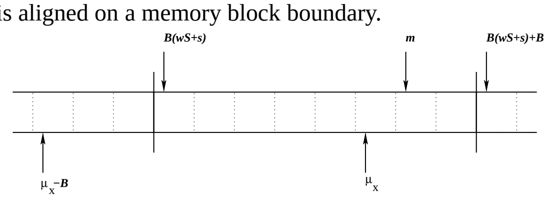
$$\begin{aligned}(R_u, \ell) \lhd (R_v, m) \stackrel{\text{def}}{=}\ & \ell \in \mathcal{I} \wedge m \in \mathcal{I} \wedge \\ & (\bigvee_{i=0}^{d-1} (\ell_i < m_i \wedge \bigwedge_{j=0}^{i-1} \ell_j = m_j) \vee \\ & (\bigwedge_{j=0}^{d-1} \ell_j = m_j \wedge u < v))\end{aligned} \tag{2}$$

3.1.3 *Mapping memory locations to cache sets*

Let A = associativity, B = block size, C = capacity, and $S = \frac{C}{AB}$ = number of cache sets. Then memory location m maps to cache set $s = \lfloor \frac{m}{B} \rfloor \bmod S$. This can be translated to the following Presburger formula, where the auxiliary variable w represents the "cache wraparound". Suppose that $Y^{(x)}$ is the array referencing memory location m, and let α_x be the number of elements in $Y^{(x)}$.

$$\begin{aligned} \text{Map}(m, w, s) &\stackrel{\text{def}}{=} 0 \leqslant s < S \wedge \\ B(wS + s) &\leqslant m < B(wS + s) + B \wedge \\ \mu_x - B &< B(wS + s) < \mu_x + \beta_x \alpha_x \end{aligned} \tag{3}$$

The last clause in formula (3) bounds the possible values of w, and is used to bound certain directions of the underlying polytope that would otherwise be unconstrained. This bounding is needed for efficiency in the counting step that follows formula simplification. The quantity $B(wS + s)$ represents the address of the first byte in the block containing memory location m, which must be within the memory locations containing array $Y^{(x)}$. However, if the starting address μ_x is not aligned on a memory block boundary, asserting that $\mu_x \leqslant B(wS + s)$ is wrong. As shown below, the address of the first byte in the memory block containing $Y^{(x)}$'s first element may actually be less than μ_x. Restricting w such that $\mu_x - B < B(wS+s)$ is correct whether or not the starting address μ_x is aligned on a memory block boundary.



3.1.4 *Data layouts in memory*

Row- and column-major layouts are easily expressed using Presburger formulas. Consider reference $R_u = (Y^{(x)}, F_u, S_h)$ and iteration point ℓ. Let $F_u(\ell) = [i_0, \ldots, i_{d_x - 1}]^T$.

$$\begin{aligned} &(m = \text{Row-maj}(F_u(\ell), \mu_x)) \stackrel{\text{def}}{=} m \geqslant 0 \\ &\wedge m = \mu_x + (\sum_{j=0}^{d_x-2} (\prod_{k=j+1}^{d_x-1} n_k) i_j + i_{d_x-1}) \beta_x \end{aligned} \tag{4}$$

$$\begin{aligned} &(m = \text{Col-maj}(F_u(\ell), \mu_x)) \stackrel{\text{def}}{=} m \geqslant 0 \\ &\wedge m = \mu_x + (i_0 + \sum_{j=1}^{d_x-1} (\prod_{k=0}^{j-1} n_k) i_j) \beta_x \end{aligned} \tag{5}$$

Section 3.6 discusses nonlinear data layouts.

3.2 Describing cache behavior using Presburger formulas

The various pieces described in Section 3.1 fit together to describe events in the cache. We now construct Presburger formulas for interior misses, boundary misses, and cache state, as defined in Sections 2.1 and 2.2. We consider direct-mapped caches for now, and extend the formulation to set-associative caches in Section 3.5.

3.2.1 *Interior misses*

To identify a cache miss, Ghosh *et al.* [26] rely on the notion of a most recent access of a memory block, which they obtain through *reuse vectors*. This abstraction is valid when the array index expressions are uniformly generated in addition to being affine in the LCVs. We avoid this condition by dispensing with the notion of a most recent access in our formulas.

To determine if an access to a memory block b results in an interior miss, it is enough to know two things: that there is an earlier access to a *different* memory block mapping to the same cache set as b; and that there is no access to b between this earlier access and the current access to b. Let reference $R_u = (Y^{(x)}, F_u, S_p)$ at iteration point i access memory block b_u, and let reference $R_v = (Y^{(y)}, F_v, S_q)$ at iteration point j access memory block b_v. Suppose that access (R_v, j) precedes access (R_u, i), recalling the "precedes" relation from Section *3.1.2*; that b_u and b_v are distinct memory blocks; but that both b_u and b_v map to the same cache set s. Then, access (R_u, i) suffers an *interior miss* if there does not exist a reference $R_w = (Y^{(z)}, F_w, S_r)$ at iteration k accessing memory block b_w, such that $(R_v, j) \lhd (R_w, k) \lhd (R_u, i)$ and $b_u = b_w$. The following formula expresses this condition.

$$\begin{aligned} ((R_u, i) \in \text{IntMiss}(\mathbb{L})) &\stackrel{\text{def}}{=} i \in \mathcal{I} \wedge \\ \exists d, s : &\text{Map}(\mathcal{L}_x(F_u(i)), d, s) \wedge \\ \exists e, j, v : &(R_v, j) \lhd (R_u, i) \wedge \\ &\text{Map}(\mathcal{L}_y(F_v(j)), e, s) \wedge \\ \neg(\exists k, w : &(R_v, j) \lhd (R_w, k) \lhd (R_u, i) \wedge \\ &\text{Map}(\mathcal{L}_z(F_w(k)), d, s)) \wedge d \neq e \end{aligned} \tag{6}$$

Note that it is not necessary to have $Y^{(z)} = Y^{(x)}$ in order to have (R_u, i) and (R_w, k) access the same memory block. This flexibility accommodates the possibility of array aliasing.

3.2.2 *Boundary misses*

Recall that boundary misses are those that are dependent on the initial cache state. Therefore, we are interested only in those accesses that are the first to map to a cache set during the execution of the loop nest. For all other accesses, the cache set already contains a memory block accessed during the execution of the loop nest, and initial cache state is irrelevant. To determine an actual boundary miss for an access that is the first to map to the cache set, it simply remains to check if the memory block accessed is resident in the initial cache state of the set.

An access $(R_u = (Y^{(x)}, F_u, S_p), i)$ to memory block b_u suffers a *boundary miss* if there does not exist an access (R_v, j) preceding (R_u, i) and accessing a memory block b_v mapping to the same cache set, and b_u is not in the initial cache state $\mathbb{C}_{in}$ at set s. Note that, unlike in the formula for interior misses, there is no constraint $b_u \neq b_v$.

$$\begin{aligned} ((R_u, i) \in \text{BoundMiss}(\mathbb{L}, \mathbb{C}_{in})) &\stackrel{\text{def}}{=} i \in \mathcal{I} \wedge \\ \exists d, s : &\text{Map}(\mathcal{L}_x(F_u(i)), d, s) \wedge \\ \neg(\exists e, j, v : &(R_v, j) \lhd (R_u, i) \wedge \\ &\text{Map}(\mathcal{L}_y(F_v(j)), e, s)) \wedge \\ &\mathcal{B}(\mathcal{L}_x(F_u(i))) \notin \mathbb{C}_{in}(s) \end{aligned} \tag{7}$$

3.2.3 *Cache state*

If the loop nest $\mathbb{L}$ contains no memory access mapping to set s, the final cache state of set s, $\mathbb{C}_{out}(s)$, is the same as the initial cache state $\mathbb{C}_{in}(s)$. Otherwise, the final cache state of set s is the address of the memory block that is not subsequently replaced by an access to a block of memory mapping to the same cache set s.

$$
\begin{aligned}
(\mathbb{C}_{out} = \Psi(\mathbb{L}, \mathbb{C}_{in})) \stackrel{\text{def}}{=} \forall s \in [0, S-1] : (\exists i : i \in \mathcal{I} \wedge \\
(\exists d : \text{Map}(\mathcal{L}_x(F_u(i)), d, s) \wedge \\
\neg(\exists e, j, v : (R_u, i) \lhd (R_v, j) \wedge \text{Map}(\mathcal{L}_y(F_v(j)), e, s)) \wedge \\
\mathbb{C}_{out}(s) = \mathcal{B}(\mathcal{L}_x(F_u(i))))) \vee \\
(\neg(\exists e : \text{Map}(\mathcal{L}_x(F_u(i)), e, s)) \wedge \mathbb{C}_{out}(s) = \mathbb{C}_{in}(s)) \quad (8)
\end{aligned}
$$

3.3 Counting cache misses

We use the Omega Calculator [33, 34] to simplify the formulas above by manipulating integer tuple relations and sets. After simplification, we are left with formulas defining a union of polytopes (see Figure 4 for an example). The number of integer points in this union is the number of misses. We use PolyLib [42] to operate on such unions. We first convert the union into a disjoint union of polytopes, and then use Ehrhart polynomials to count the number of integer points [18] in each polytope.

3.4 Extension to imperfect loop nests

Extending our model to imperfect loop nests involves two steps.

1. We use the transformations of Ahmed *et al.* [2, 3] to convert an imperfect loop nest into a perfect loop nest with guards on statements.

2. We extend the notion of a *valid iteration point* to that of a *valid access*.

For each statement of the loop nest, Ahmed *et al.* define a *statement iteration space* whose dimension is the number of loops that contain the statement. The *product space* for the loop nest is a linearly independent subspace of the the Cartesian product of all the statement iteration spaces. Affine *embedding functions* map a point in a statement iteration space to a point in the product space. When multiple statements map to the same iteration point in product space, they are executed in program order. In relation to the product space, embeddings represent guards on statements, mapping a statement from its place outside the innermost loop to a valid place inside the innermost loop. We emphasize that the guards are conceptual, and for analysis only. They do not result in run-time conditional tests in the generated code.

Kelly and Pugh [35, 36] and Lim and Lam [41] have presented other algorithms that embed imperfect loop nests into perfect loop nests, with similar end results. The details of the embedding algorithms are not important for our purpose. Our use of the framework of Ahmed *et al.* merely reflects our greater familiarity with their work.

Figure 1(a) is an improved version of Example 1, in which the loop-invariant reference `C[i,j]` is hoisted out of the `k`-loop and stored in a scalar `x` that can be register-resident. In this imperfect loop nest, statements `S0` and `S2` occur outside of the innermost loop. Let iX denote the loop index variable i pertaining to statement `SX`. Then $i0 \times j0$ and $i2 \times j2$ are the statement iteration spaces of statements `S0` and `S2`, respectively. The following embedding functions

$$
F0(\begin{bmatrix} i0 \\ j0 \end{bmatrix}) = \begin{bmatrix} i0 \\ j0 \\ 0 \end{bmatrix}, \quad F2(\begin{bmatrix} i2 \\ j2 \end{bmatrix}) = \begin{bmatrix} i2 \\ j2 \\ n-1 \end{bmatrix},
$$

map points in these statement iteration spaces to points in product space $[i, j, k]^T$. It is clear how the guards on statements `S0` and `S2` of Figure 1(b) accomplish this. Statement `S1` is already in the innermost loop, and requires no guard on it.

The second part of the extension is to insure that our model can handle array references that are guarded in this manner. We accomplish this effect by extending our notion of a *valid iteration point* (Section 3.1) to that of a *valid access*.

Let $R_u = (Y^{(x)}, F_u, S_h)$ be the u^{th} reference with $0 \leqslant u < k$. Let $G_h(i)$ be the *guard* of statement S_h in the product space version of the loop nest. We assume that the guards are expressible in Presburger arithmetic. For Figure 1(b), $G_0 = (i_2 = 0)$, $G_1 =$ **true**, and $G_2 = (i_2 = n_2 - 1)$. Then $(R_u = (Y^{(x)}, F_u, S_h), i)$ is a *valid access* if i belongs to the iteration space, and $G_h(i)$ holds. The predicate $(R_u, i) \bar{\in} \mathcal{I}$ represents this fact,

$$
(R_u, i) \bar{\in} \mathcal{I} \stackrel{\text{def}}{=} i \in \mathcal{I} \wedge 0 \leqslant u < k \wedge G_h(i) \quad (9)
$$

With this extension, the formulas from Section 3.2 apply directly, with every occurrence of $i \in \mathcal{I}$ replaced by $(R_u, i) \bar{\in} \mathcal{I}$.

3.5 Associativity

We currently handle associativity in a straightforward manner, assuming a Least Recently Used replacement policy. From Section *3.2.1*, we simply need to allow at least A distinct accesses preceding (R_u, i) to unique memory blocks, such that there is no access (R_w, k) accessing the same memory block as (R_u, i) and $(R_{v_0}, j_0) \lhd (R_w, k) \lhd (R_u, i)$ (where (R_{v_0}, j_0) is the earliest of at least A references to unique memory blocks). The following Presburger formula expresses interior misses for an A-way set-associative cache.

$$
\begin{aligned}
((R_u, i) \in \text{IntMiss}) \stackrel{\text{def}}{=} i \in \mathcal{I} \wedge \\
\exists d, s : \text{Map}(\mathcal{L}_x(F_u(i)), d, s) \wedge \\
\exists e_0, j_0, v_0 : (R_{v_0}, j_0) \lhd (R_u, i) \wedge \\
\text{Map}(\mathcal{L}_{y_0}(F_{v_0}(j_0)), e_0, s) \wedge \\
(\exists e_1, \ldots, e_{A-1} : \\
\bigwedge_{a=1}^{A-1} (\exists j_a, v_a : (R_{v_0}, j_0) \lhd (R_{v_a}, j_a) \lhd (R_u, i) \wedge \\
\text{Map}(\mathcal{L}_{y_a}(F_{v_a}(j_a)), e_a, s)) \wedge \\
d \neq e_0 \neq \cdots \neq e_{A-1}) \wedge \\
\neg(\exists k, w : (R_{v_0}, j_0) \lhd (R_w, k) \lhd (R_u, i) \wedge \\
\text{Map}(\mathcal{L}_z(F_w(k)), d, s)) \quad (10)
\end{aligned}
$$

This method will handle modest values of A, and the complexity of the formulas certainly increases with A. Presburger formulas for cache state and boundary misses with associativity A are non-obvious, and will require more work to construct.

3.6 Array layouts based on bit interleaving

Previous work [14, 15, 21] suggests that non-linear data layouts provide better cache performance than canonical layout functions in some numerical codes. Such layout functions are described in terms of interleavings of the bits in the binary expansions of the array coordinates rather than as affine functions of the numerical values of these quantities. We describe such bit interleavings and provide formulations of these layouts in Presburger arithmetic.

In developing the model of alternative array layouts, we assume that $n_j = 2^{q_j}$ for some $j \in [0, d_x - 1]$ (where d_x is the number of coordinates in an array $Y^{(x)}$). Therefore, the bit representation of an array index will have q_j bits, with the least significant bit (LSB) numbered 0 and the most significant bit (MSB) numbered $q_j - 1$. We identify the binary sequence $s_{q-1} \ldots s_0$ with the non-

```
      do i = 0, n-1
        do j = 0, n-1
S0:       x = C[i,j]
          do k = 0, n-1
S1:         x = A[i,k]*B[k,j] + x
          end
S2:       C[i,j] = x
        end
      end
```

```
      do i = 0, n-1
        do j = 0, n-1
          do k = 0, n-1
S0:         if (k == 0) x = C[i,j]
S1:         x = A[i,k]*B[k,j] + x
S2:         if (k == n-1) C[i,j] = x
          end
        end
      end
```

Figure 1: (a) An imperfect loop nest for matrix multiplication. (b) The product space version with guards.

negative integer $s = \sum_{i=0}^{q_j-1} s_i 2^i$. We denote by B_{q_j} the set of all binary sequences of length q_j, and extend the above identification to identify B_{q_j} with the interval $[0, 2^{q_j} - 1]$.

We describe a family of nonlinear layout functions parameterized by a single parameter σ, as follows. An $(q_0, \ldots, q_{d_x-1})$-*interleaving*, σ, is a sequence of length p (where $p = \sum_{i=0}^{d_x-1} q_i$) over the alphabet $\{0, \ldots, (d_x - 1)\}$ containing q_i i's. It describes the order in which bits from the d_x array coordinates are interleaved to linearize the array in memory.

An array layout functions as a map from d_x array coordinates to a memory address. Therefore, given an $(q_0, \ldots, q_{d_x-1})$-interleaving σ, define a map

$$\Theta : B_{q_0} \times \cdots \times B_{q_{d_x-1}} \to B_p$$

in the following way. If $x^{(i)} = x^{(i)}_{q_i-1} \ldots x^{(i)}_1 x^{(i)}_0 \in B_{q_i} \forall i \in [0, d_x - 1]$, then $\Theta(x^{(0)}, \ldots, x^{(d_x-1)})$ is the sequence obtained by replacing the jth u from the right with $x^{(u)}_j$. We extend this notation to consider Θ as a map from $[0, 2^{q_0} - 1] \times \cdots \times [0, 2^{q_{d_x-1}} - 1]$ to $[0, 2^p - 1]$ by identifying non-negative integers and their binary expansions. We call Θ the *mixing function* indexed by σ. Note that $\Theta(0, \ldots, 0) = 0$ for any σ.

Example 2 Let $d_x = 2$, $n_0 = 16$ ($q_0 = 4$), $n_1 = 16$ ($q_1 = 4$), and $\sigma = 10110010$. Then

$$\Theta(12, 5) = \Theta(1100, 0101) = 01101010 = 106.$$

Example 3 Let $d_x = 3$, $n_0 = 8$ ($q_0 = 3$), $n_1 = 8$ ($q_1 = 3$), $n_2 = 4$ ($q_2 = 2$), and let $\sigma = 21102001$. Then

$$\Theta(3, 7, 1) = \Theta(011, 111, 001) = 01101111 = 111.$$

The idea behind translating such a data layout into a Presburger formula is to define the bit values of the binary expansion of the memory address using Presburger arithmetic. Consider again reference $R_u = (Y^{(x)}, F_u, S_h)$ and iteration point ℓ. For every n_j where $0 \leqslant j < d_x$, let $n_j = 2^{q_j}$. Then σ is an $(q_0, \ldots, q_{d_x-1})$-interleaving. Then we can compute the following $d_x \times p$ matrix $\mathbb{Z}(\sigma)$. Letting $g = \sigma_f$, the f^{th} column of $\mathbb{Z}(\sigma)$ consists of 2^e in the g^{th} position, where σ_f is the e^{th} g from the right, and zeros in every other position. $\mathbb{Z}(\sigma)$ can be thought of as a transformation that when applied to the binary expansion of a memory address m, produces the coordinates of the array element at m.

Example 4 Given that $d_x = 3$, $n_0 = 8$ ($q_0 = 3$), $n_1 = 8$ ($q_1 = 3$), $n_2 = 4$ ($q_2 = 2$), and $\sigma = 12102010$,

$$\mathbb{Z}(\sigma) = \begin{bmatrix} 0 & 0 & 0 & 4 & 0 & 2 & 0 & 1 \\ 4 & 0 & 2 & 0 & 0 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

The following formula maps $F_u(\ell)$, μ_x, and $\mathbb{Z}(\sigma)$ to memory location m. Let $p = \sum_{j=0}^{d_x-1} q_j$, and let $M = [m_{p-1}, \cdots, m_0]^T$.

$$\begin{aligned} (m = \text{Interleave}(F_u(\ell), \mu_x, \mathbb{Z}(\sigma))) &\stackrel{\text{def}}{=} \\ \exists m_{off}, m_{p-1}, \ldots, m_0 &: \\ 0 \leqslant m_{p-1}, \ldots, m_0 \leqslant 1 \wedge m \geqslant 0 &\wedge \\ m = \mu_x + m_{off}\beta_x \wedge F_u(\ell) = \mathbb{Z}(\sigma) M &\wedge \\ m_{off} = \sum_{k=0}^{p-1} m_k 2^k & \end{aligned} \quad (11)$$

Data layouts such as X-Morton and U-Morton [15] require an X-OR operation in addition to bit interleaving. (Note that this formalism applies only to $n \times n$ arrays.) The additional X-OR operation can also be expressed as a Presburger formula on the bit representation.

3.7 Physically addressed caches

The techniques described thus far operate on virtual addresses. However, many systems utilize physical indexed caches (*e.g.*, second level caches) whose performance is highly dependent on page placement. Fortunately, most operating systems employ page coloring techniques that minimize this effect [37] by creating virtual to physical page mappings such that the virtual and physical cache index are identical. It may also be possible to extend our analysis to include the effects of page placement; we leave this as future research.

4. RESULTS

In this section, we present and interpret cache behavior as obtained by our method on five model problems, and validate them against cache miss counts produced by a (specially-written) cache simulator. Unless otherwise specified, we use a direct-mapped cache with capacity 4096 bytes and block size of 32 bytes that is initially empty. We assume that all data arrays contain double-precision numbers (so that β is eight bytes), and that all arrays are linearized in column-major order. The total number of misses for each array match up exactly between our model and the simulator in all cases, but their partitioning differs. We explain the implications of this difference in Section 4.1.

Problem 1 (Matrix multiplication) We count boundary and interior misses for each array for the matrix multiplication kernel shown in Example 1, under four scenarios.

1. Problem size $n = 21$, the leading dimension of each array is n, and the three arrays are adjacent to each other in memory address space (*i.e.*, $\mu_A = 0$, $\mu_B = \beta n^2$, and $\mu_C = 2\beta n^2$). We show results for all six possible permutations of the loop

orders, from both our approach and from explicit cache simulation. This is representative of a code where both the iteration space and the data arrays are tiled. Placing the arrays back-to-back causes two memory blocks to be shared between arrays. Figure 2(a) tabulates the results. The `jki` loop order is seen to be substantially superior in terms of total misses.

2. Problem size $n = 20$, the leading dimension of each array is n, and the three arrays are adjacent to each other in memory address space. We show results for all six possible permutations of the loop orders, from both our approach and from explicit cache simulation. This scenario is similar to the previous one, but there is no sharing of memory blocks between arrays. Figure 2(b) tabulates the results. The number of misses is somewhat smaller, and the `jki` loop order wins again.

3. Problem size $n = 21$, the leading dimension of each array is n, and the three arrays collide in cache space (*i.e.*, $\mu_A = 0$, $\mu_B = 4096$, and $\mu_C = 8192$). This represents a situation where the arrays do not use the cache effectively (occupying only 111 of the 128 cache sets). We show results for all six possible permutations of the loop orders, from both our approach and from explicit cache simulation. Figure 2(c) tabulates the results. The number of misses rises dramatically, as expected; the `jki` loop order produces the fewest cache misses, but not by as large a margin.

4. Problem size $n = 20$, the leading dimension of each array is kn (for $k \in \{1, 2, 3\}$), and the three arrays are adjacent to each other in memory address space. This represents a situation where the iteration space is tiled but the data is not reorganized, resulting in the data tiles not being contiguous in memory space. We show only the `ijk` loop order. Figure 2(d) tabulates the results. The total number of misses for each array change with the leading dimension, although different arrays behave differently.

Problem 2 (Multiple loop nests) We count boundary and interior misses for each array for the following variation on the matrix multiplication kernel.

```
do i = 0, n-1        /* Loop nest 1 */
  do j = 0, n-1
    C[i,j] = 0
  end
end
do i = 0, n-1        /* Loop nest 2 */
  do j = 0, n-1
    do k = 0, n-1
      C[i,j] = A[i,k]*B[k,j] + C[i,j]
    end
  end
end
```

The layout constraints are identical to those in Problem 1, scenario 1. This demonstrates how the model handles multiple loop nests.

The miss counts are as follows.

	A			B			C		
Loop	Bnd	Int	Tot	Bnd	Int	Tot	Bnd	Int	Tot
1	0	0	0	0	0	0	111	0	111
2	28	521	549	92	866	958	0	383	383

The model correctly classifies all the misses in the first loop nest as boundary misses. The cache contains all of array C at the end of the first loop nest, so all of the misses of C in the second loop nest are interior misses. Figure 3 graphically represents cache state at the end of the computation.

Problem 3 (Imperfect loop nest) We count boundary and interior misses for each array for the imperfect loop version of the matrix multiplication kernel of Figure 1 with $n = 21$, with the leading dimension of each array being n. This demonstrates how the model handles imperfect loop nests. We show two scenarios.

The first scenario has the three arrays adjacent to each other in memory address space. The miss counts are as follows.

	A	B	C (read)	C (write)
Bnd	28	92	8	0
Int	521	866	383	0
Total	549	958	391	0
Cold	110	110	111	0
Repl	439	848	280	0

The significant observation is that none of the write references to C miss, even though there are many references to A and B between the read and the write reference to `C[i,j]`. The total number of misses is identical to that of Problem 1, scenario 1.

The second scenario has the arrays colliding in the cache. The miss counts are as follows.

	A	B	C (read)	C (write)
Bnd	20	90	1	0
Int	980	648	440	441
Total	1000	738	441	441
Cold	111	111	111	0
Repl	889	627	330	441

Now every read and write reference to `C[i,j]` misses. However, the total number of cache misses is significantly smaller than the corresponding case in Problem 1, scenario 3, showing the benefit of allocating `C[i,j]` in a register.

Problem 4 (Set-associative cache) We count interior misses for each array for the matrix multiplication kernel shown in Example 1, using two-way associative caches. The layout constraints are identical to those in Problem 1, scenario 2. This demonstrates how the model handles associativity.

Both scenarios consider a two-way associative cache with block size of 32 bytes that is initally empty. The cache has a capacity of 4096 bytes in the first scenario and 8192 bytes in the second scenario. The miss counts are as follows.

	$C = 4096$			$C = 8192$		
	A	B	C	A	B	C
Bnd		128			256	
Int	75	773	213	8	0	36
Total		1189			300	
Cold	100	100	100	100	100	100
Repl	0	757	132	0	0	0

The total number of boundary misses in each scenario is determined by the number of cache frames in the footprint of all three arrays in cache. For every cache frame that is touched during the matrix multiplication kernel, the first instance of a memory block being mapped to the cache frame incurs a boundary miss since the cache is initially empty. In the first scenario, there are 64 cache

(a)

Loop order	A Bnd	A Int	A Tot	A Cold	A Repl	B Bnd	B Int	B Tot	B Cold	B Repl	C Bnd	C Int	C Tot	C Cold	C Repl	Grand Total
ijk	28	521	549	110	439	92	866	958	110	848	8	383	391	111	280	1898
ikj	18	445	463	110	353	85	1985	2070	110	1960	25	1563	1588	111	1477	4121
jik	108	590	698	110	588	18	502	520	110	410	2	109	111	111	0	1329
jki	104	355	459	110	349	18	167	185	110	75	6	207	213	111	102	857
kij	2	184	186	110	76	34	1644	1678	110	1568	92	1624	1716	111	1605	3580
kji	9	297	306	110	196	31	436	467	110	357	88	530	618	111	507	1391

(b)

Loop order	A Bnd	A Int	A Tot	A Cold	A Repl	B Bnd	B Int	B Tot	B Cold	B Repl	C Bnd	C Int	C Tot	C Cold	C Repl	Grand Total
ijk	25	405	430	100	330	85	661	746	100	646	18	298	316	100	216	1492
ikj	23	349	372	100	272	73	1533	1606	100	1506	32	1205	1237	100	1137	3215
jik	97	409	506	100	406	28	345	373	100	273	3	97	100	100	0	979
jki	95	261	356	100	256	28	131	159	100	59	5	160	165	100	65	680
kij	13	146	159	100	59	33	1276	1309	100	1209	82	1254	1336	100	1236	2804
kji	16	220	236	100	136	31	352	383	100	283	81	404	485	100	385	1104

(c)

Loop order	A Bnd	A Int	A Tot	A Cold	A Repl	B Bnd	B Int	B Tot	B Cold	B Repl	C Bnd	C Int	C Tot	C Cold	C Repl	Grand Total
ijk	21	964	985	111	874	90	1799	1889	111	1778	0	2393	2393	111	2282	5267
ikj	1	864	865	111	754	110	1846	1956	111	1845	0	2556	2556	111	2445	5377
jik	107	578	685	111	574	4	1900	1904	111	1793	0	2123	2123	111	2012	4712
jki	111	558	669	111	558	0	1789	1789	111	1678	0	2232	2232	111	2121	4690
kij	5	545	550	111	439	20	1866	1886	111	1775	86	2299	2385	111	2274	4821
kji	6	577	583	111	472	5	1823	1828	111	1717	100	2229	2329	111	2218	4740

(d)

k	A Bnd	A Int	A Tot	A Cold	A Repl	B Bnd	B Int	B Tot	B Cold	B Repl	C Bnd	C Int	C Tot	C Cold	C Repl	Grand Total
1	25	405	430	100	330	85	661	746	100	646	18	298	316	100	216	1492
2	40	305	345	100	245	71	1198	1269	100	1169	17	227	244	100	144	3350
3	40	449	489	100	389	68	1119	1187	100	1087	20	311	331	100	231	5357

Figure 2: Miss counts from our approach (Bnd and Int) and from cache simulation (Cold and Repl). (a) Problem 1, scenario 1. (b) Problem 1, scenario 2. (c) Problem 1, scenario 3. (d) Problem 1, scenario 4.

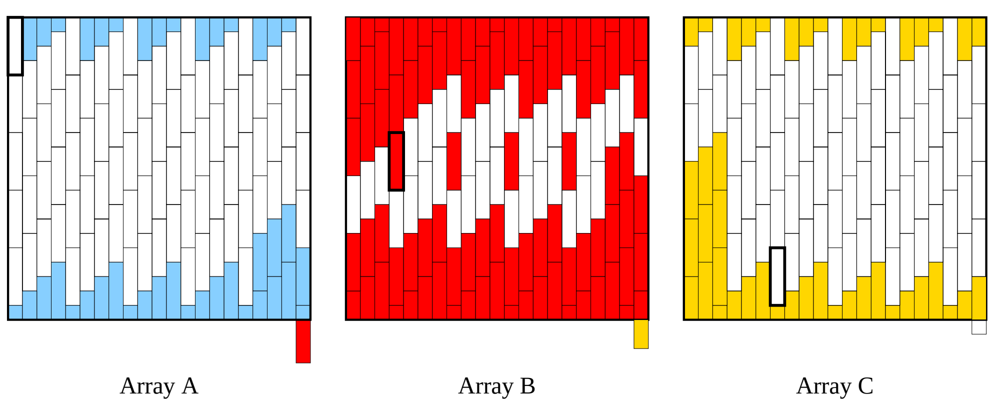


Figure 3: Cache state at the end of the computation described in Problem 2. The shaded blocks are cache-resident. There are exactly 128 shaded memory blocks. Arrays A and B share a block, as do arrays B and C. The block with the heavy outline in each array maps to cache set 0.

sets. Since the cache footprint of all arrays 'wraps around' at least twice, we know that all 128 cache frames are touched. Hence, there are 128 boundary misses in scenario 1. Similarly, we can determine that there are 256 boundary misses in scenario 2.

Problem 5 (Symbolic analysis) We analyze the matrix-vector product example from Fricker *et al.* [22] to show the symbolic processing capabilities of our approach. The code is as follows.

```
do j1 = 0, N-1
   reg = Y[j1]
   do j2 = 0, N-1
      reg += A[j2, j1] * X[j2]
   end
   Y[j1] = reg
end
```

We focus on the interior misses on X due to interference from A, assuming that $\mu_A = 0$ but that μ_X is symbolic. For compatibility with Fricker *et al.*, we use a direct-mapped cache of capacity 8192 bytes and block size of 32 bytes, and we choose $N = 100$. The formula shown in Figure 4 is a pretty-printed version of formula (6) as simplified by the Omega Calculator. While the formula appears formidable, it should be kept in mind that it captures the miss patterns for all possible values of μ_X. In principle, the Ehrhart polynomial of the polytope union represented by this formula can be computed, enabling counting of the number of misses for a particular value of μ_X by evaluating this polynomial.

4.1 Interpretation of results

Two general observations on the results are worth mentioning.

First, the model results are identical to the simulation results in all cases. This reinforces the exactness of the model, which is a major strength.

Second, the classification of cache misses into *boundary* and *interior* misses rather than *cold* and *replacement* misses is a significant departure from previous models. Boundary misses are, in a sense, a cache-centric analog of cold misses. Just as the number of cold misses of a program is no more than the number of memory blocks occupied by the data, the number of boundary misses is no more than the combined cache footprint of the data, which is itself bounded above by the number of cache sets. This can be verified by totaling the boundary or cold misses in any row of Figure 2(a) (where the totals are 128 and 300, respectively) or Figure 2(b) (where the totals are 111 and 333, respectively). In general, then, the number of boundary misses should be significantly smaller than the number of cold misses. Thus, our classification allows for more precise context-free identification of misses, leaving many fewer references to be resolved from cache state.

4.2 Running times

Figure 5 shows a histogram of the running times required by the Omega Calculator [33] to simplify 108 sample cache miss formulas on a 300MHz Sparc Ultra 60. The samples are made up of boundary and interior miss formulas for each array in scenarios 1, 2, and 3 of Problem 1. The cache miss formulas are generated from source code using a SUIF [55] compiler pass that we developed for this purpose. The time required for formula generation is negligible.

Half of the cache miss formulas run in less than 10 seconds, and the majority of those formulas run in less than 1 second. The boundary miss formulas simplified quickly (most in under a second), while the time required to simplify the interior miss formulas varied widely. We have observed the running time of a formula to be strongly correlated to the number of cache misses it generates.

Given these running times, our approach clearly does not yet have enough performance to be practical; however, the value of in-

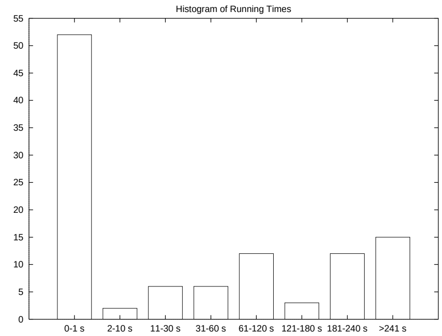


Figure 5: Histogram of running times of formula simplification using the Omega calculator on a 300 MHz Sparc Ultra 60.

sight gained from our approach should not be overlooked. Furthermore, it is not clear how much of the slow running time is a consequence of our formulations and how much is due to the Omega software. We hope to make this determination in the immediate future, by investigating other software options.

5. RELATED WORK

We organize related work by the way in which they handle cache behavior: compiler-centric, language-centric, architecture-centric, or trace-centric (including simulation).

Compiler-centric. The work of Ghosh *et al.* [23, 24, 25, 26] is most closely related to our framework for analytical cache modeling. (Zhang and Martonosi [64] have recently begun extending this work to pointer data structures.) They introduce additional constraints to make the problem tractable. We avoid these constraints.

Work by Ahmed *et al.* on tiling imperfect loop nests [2, 3] embeds the iteration space of each statement of a loop into a product space. We use this transformation in Section 3.4.

Caşcaval [12] estimates the number of cache misses using *stack distances*. His prediction is valid for a fully-associative cache with LRU replacement, and requires a probabilistic argument to transfer to a cache with smaller associativity. He assumes that each loop nest starts with a cold cache, sacrificing the accuracy gained from knowing the actual cache state at the start of the loop nest.

Our simplified formulas resemble LMADs [47] in several respects. Establishing a connection between them remains the subject of future work.

Language-centric. Alt *et al.* [5] apply *Abstract Interpretation* to predicting the behavior of instruction cache, for general programs. Their notion of cache state is somewhat different from ours.

Prior empirical evidence [14, 15, 21] suggests that alternative array layout functions provide better cache behavior than canonical layout functions for many dense linear algebra codes. Previous work [27] has taken a combinatorial approach to modeling cache misses in the presence of such non-linear data layouts.

Architecture-centric. Lam, Rothberg, and Wolf [38] discuss the importance of cache optimizations for blocked algorithms, using matrix multiplication as an example. Their simulation-based analysis is exact, but their performance models are approximate.

Fricker *et al.* [22] develop a model for approximating cache interferences incurred while executing blocked matrix vector multiply in a specific cache. Their analysis is inexact in considering only cross-interferences and neglecting redundancies among array pairs.

McKinley and Temam [44] examine locality characteristics in the SPEC'95 and Perfect Benchmarks. Their discovery most perti-

$$
\begin{aligned}
&\{[j_1, 0, s, d] : \exists(\alpha : 1 \leqslant j_1 \leqslant 99 \wedge 0 \leqslant s \leqslant 255 \wedge \alpha < d \wedge 32s + 8192d \leqslant \mu_X \wedge 800j_1 + 8192d \leqslant 792 + \mu_X + 8192\alpha \wedge \\
&\qquad \mu_X \leqslant 31 + 32s + 8192d \wedge s + 256\alpha < 25j_1)\} \\
&\cup\{[j_1, j_2, s, d] : \exists(\alpha : 1 \leqslant j_1 \leqslant 99 \wedge 0 \leqslant s \leqslant 255 \wedge 1 \leqslant j_2 \wedge 925 + 100j_1 + j_2 \leqslant 1024d + 4s \wedge 8192d + 32s \leqslant \mu_X + 8j_2 \wedge \\
&\qquad \mu_X + 8j_2 \leqslant 7 + 8192d + 32s \wedge 100j_1 + j_2 \leqslant 99 + 4s + 1024\alpha \wedge s + 256\alpha < 25j_1)\} \\
&\cup\{[j_1, j_2, s, d] : \exists(\alpha : 0 \leqslant j_1 \leqslant 99 \wedge 0 \leqslant s \leqslant 255 \wedge j_2 \leqslant 99 \wedge 25j_1 \leqslant s + 256\alpha \wedge 8192d + 32s \leqslant \mu_X + 8j_2 \wedge \\
&\qquad \mu_X + 8j_2 \leqslant 7 + 8192d + 32s \wedge 4s + 1024\alpha < 100j_1 + j_2 \wedge 256 + 25j_1 \leqslant 256d + s)\} \\
&\cup\{[j_1, j_2, s, d] : \exists(\alpha : 0 \leqslant j_1 \leqslant 99 \wedge 0 \leqslant j_2 \leqslant 99 \wedge 0 \leqslant s \leqslant 255 \wedge 8192d + 32s \leqslant \mu_X + 8j_2 \wedge \mu_X + 8j_2 \leqslant 31 + 8192d + 32s \wedge \\
&\qquad 1021 + 100j_1 + j_2 \leqslant 1024d + 4s \wedge 100j_1 + j_2 \leqslant 3 + 4s + 1024\alpha \wedge 4s + 1024\alpha \leqslant 100j_1 + j_2)\} \\
&\cup\{[j_1, 0, s, d] : \exists(\alpha : 1 \leqslant j_1 \leqslant 99 \wedge 0 \leqslant s \leqslant 255 \wedge 257 + s + 256d \leqslant 25j_1 \wedge 32s + 8192d \leqslant \mu_X \wedge \\
&\qquad 800j_1 + 8192d \leqslant 792 + \mu_X + 8192\alpha \wedge \mu_X \leqslant 31 + 32s + 8192d \wedge s + 256\alpha < 25j_1)\} \\
&\cup\{[j_1, j_2, s, d] : \exists(\alpha : 1 \leqslant j_1 \leqslant 99 \wedge 0 \leqslant s \leqslant 255 \wedge 1 \leqslant j_2 \wedge 257 + 256d + s \leqslant 25j_1 \wedge 8192d + 32s \leqslant \mu_X + 8j_2 \wedge \\
&\qquad \mu_X + 8j_2 \leqslant 7 + 8192d + 32s \wedge 100j_1 + j_2 \leqslant 99 + 4s + 1024\alpha \wedge s + 256\alpha < 25j_1)\} \\
&\cup\{[j_1, j_2, s, d] : \exists(\alpha : 0 \leqslant j_1 \leqslant 99 \wedge 0 \leqslant s \leqslant 255 \wedge j_2 \leqslant 99 \wedge 25j_1 \leqslant s + 256\alpha \wedge 8192d + 32s \leqslant \mu_X + 8j_2 \wedge \\
&\qquad \mu_X + 8j_2 \leqslant 7 + 8192d + 32s \wedge 4s + 1024\alpha < 100j_1 + j_2 \wedge 1025 + 1024d + 4s \leqslant 100j_1 + j_2)\} \\
&\cup\{[j_1, j_2, s, d] : \exists(\alpha : 0 \leqslant j_1 \leqslant 99 \wedge 0 \leqslant j_2 \leqslant 99 \wedge 0 \leqslant s \leqslant 255 \wedge 8192d + 32s \leqslant \mu_X + 8j_2 \wedge \mu_X + 8j_2 \leqslant 31 + 8192d + 32s \wedge \\
&\qquad 1024 + 1024d + 4s \leqslant 100j_1 + j_2 \wedge 100j_1 + j_2 \leqslant 3 + 4s + 1024\alpha \wedge 4s + 1024\alpha \leqslant 100j_1 + j_2)\}
\end{aligned}
$$

Figure 4: A formula describing interior misses on X due to interference from A in Problem 5. Each 4-tuple is of the form $[j_1, j_2, s, d]$, where (j_1, j_2) identifies the iteration at which the miss occurs, and (s, d) identifies the set and the cache wraparound of the reference.

nent to our work is that most misses are *internest capacity misses*.

Harper *et al.* [28] present an analytical model that focuses on set-associative caches. Their model approximates the cache miss-ratio of a looping construct and allows imperfect loop nests to be considered. They do not attempt to analyze multiple loop nests.

Trace-centric. Prior research [1, 57] has investigated various analytic cache models by extracting parameters from the reference trace. Simulation techniques, such as cache profiling [43, 39], can provide insight on potential program transformations by classifying misses according the cause of the cache miss. All trace-centric approaches usually require full execution of the program.

Weikle *et al.* [58, 59] introduce the novel idea of viewing *caches as filters*. This framework is not limited to analyzing loop nests or other particular program constructs, but can handle any pattern of memory references. Brehob and Enbody [11] model locality using distances between memory references in a trace.

Wood *et al.* [63] explore the problem of resolving *unknown references* in simulation—first-time references to memory blocks that may miss or hit depending on the cache state at the beginning of the trace sample—and show that accurate estimation of their miss rate is necessary. We use cache state to resolve such unknown references, and then categorize them as boundary misses or hits.

6. CONCLUSIONS AND FUTURE WORK

This work initially began from the intuition that the CME formulation of Ghosh *et al.* was not fully exploiting all of the regularity inherent in the problem. The output of the Presburger formulas vividly illustrates this regularity, allowing us to employ general-purpose tools for counting misses.

While powerful mathematical results (such as the existence of the Ehrhart polynomial) are known for polytopes, the corresponding algorithms are complex and subject to geometric degeneracies. As a result, the software libraries are not very robust. Such degeneracies have prevented us, for example, from calculating the Ehrhart polynomial for the formula in Figure 4. Similar comments apply, with less severity, to the Presburger decision procedures. The robustness of both libraries needs to be improved substantially to realize the full potential of our approach.

While we have made some progress in handling associativity, symbolic constants, and non-linear array layouts, much remains to be done on all three fronts. Our current handling of associativity is incomplete and unscalable; in particular, it is not powerful enough to model TLB behavior. Our ability to handle symbolic constants derives from, and is therefore limited by, the corresponding capability in Omega. The constraints introduced in Section 3.6 to handle non-linear data layouts are essentially 0–1 integer programming constraints, which are likely to cause bad behavior in the Presburger decision procedures.

We have recently become aware of an alternative tool [10] that claims to be more aggressive at formula simplification than Omega, and also of an alternative approach to representing Presburger formulas using finite automata [8, 9]. We intend to explore both these options to try to improve the efficiency of our system. However, the general problem of simplifying arbitrary Presburger formulas is intrinsically difficult, no matter whether one views it from the perspective of logic, number theory, computational geometry [53], automata theory, or something else. In the end, the only practical path to efficiency may involve developing specialized algorithms that exploit some structural constraints of the kinds of formulas that arise in our application.

In addition to compiler-related uses, our approach may also significantly speed up cache simulators by enabling them to rapidly leap-frog the computation over polyhedral loop nests that consume most of the running time. The development of such a mixed-mode cache simulator remains the subject of future work.

7. REFERENCES

[1] A. Agarwal, M. Horowitz, and J. Hennessy. An analytical cache model. *ACM Trans. Comput. Syst.*, 7(2):184–215, May 1989.

[2] N. Ahmed. *Locality Enhancement of Imperfectly-nested Loop Nests*. PhD thesis, Department of Computer Science, Cornell University, Aug. 2000.

[3] N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loop nests. Technical Report TR2000-1782, Cornell University, 2000.

[4] N. Ahmed and K. Pingali. Automatic generation of block-recursive codes. In *Proceedings of Europar 2000*, pages 125–134, 2000.

[5] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In R. Cousot and D. A. Schmidt, editors, *SAS'96, Static Analysis Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 51–66. Springer, September 1996.

[6] T. Amon, G. Borriello, T. Hu, and J. Liu. Symbolic timing verification of timing diagrams using Presburger formulas. In *Proceedings of DAC 97*, pages 226–231, Anaheim, CA, June 1997.

[7] T. Amon, G. Borriello, and J. Liu. Making complex timing relationships readable: Presburger formula simplification using don't cares. In *Proceedings of DAC 98*, pages 586–590, San Francisco, CA, June 1998.

[8] B. Boigelot and P. Wolper. An automata-theoretic approach to Presburger arithmetic. In A. Mycroft, editor, *Proceedings of the Second International Symposium on Static Analysis (SAS '95)*, volume 983 of *Lecture Notes in Computer Science*, pages 1–18. Springer Verlag, Sept. 1995.

[9] A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In H. Kirchner, editor, *Proc. Coll. on Trees in Algebra and Programming (CAAP'96)*, volume 1059 of *Lecture Notes in Computer Science*, pages 30–43. Springer Verlag, 1996.

[10] P. Boulet and X. Redon. SPPoC: fonctionnemen et applications. Research Report 00-04, LIFL (Laboratoire de Recherche en Informatique de l'Université des Sciences et Technologies de Lille), 2000. In French. Also see `http://www.lifl.fr/west/sppoc/`.

[11] M. Brehob and R. Enbody. A mathematical model of locality and caching. Technical Report TR-MSU-CPS-96-TBD, Michigan State University, Nov. 1996.

[12] G. C. Caşcaval. *Compile-Time Performance Prediction of Scientific Programs*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2000.

[13] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, San Jose, CA, Oct. 1994.

[14] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, pages 444–453, Rhodes, Greece, June 1999.

[15] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi. Recursive array layouts and fast parallel matrix multiplication. In *Proceedings of Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 222–231, Saint-Malo, France, June 1999.

[16] S. Chatterjee and S. Sen. Cache-efficient matrix transposition. In *Proceedings of HPCA-6*, pages 195–205, Toulouse, France, Jan. 2000.

[17] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 205–217, La Jolla, CA, June 1995.

[18] P. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In *Proceedings of International Conference on Supercomputing*, pages 278–285, May 1996.

[19] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 279–290, La Jolla, CA, June 1995.

[20] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–54, 1991.

[21] J. D. Frens and D. S. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance with source code. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 206–216, Las Vegas, NV, June 1997.

[22] C. Fricker, O. Temam, and W. Jalby. Influence of cross-interference on blocked loops: A case study with matrix-vector multiply. *ACM Trans. Prog. Lang. Syst.*, 17(4):561–575, July 1995.

[23] S. Ghosh. *Cache Miss Equations: Compiler analysis framework for tuning memory behavior*. PhD thesis, Department of Electrical Engineering, Princeton University, Nov. 1999.

[24] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 1997 International Conference on Supercomputing*, pages 317–324, Vienna, Austria, July 1997.

[25] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 228–239, San Jose, CA, Oct. 1998.

[26] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Trans. Prog. Lang. Syst.*, 21(4):703–746, July 1999.

[27] P. J. Hanlon, D. Chung, S. Chatterjee, D. Genius, A. R. Lebeck, and E. Parker. The combinatorics of cache misses during matrix multiplication. *J. Comput. Syst. Sci.*, 2000. To appear.

[28] J. S. Harper, D. J. Kerbyson, and G. R. Nudd. Analytical modeling of set-associative cache behavior. *IEEE Trans. Comput.*, 48(10):1009–1024, Oct. 1999.

[29] M. D. Hill, J. R. Larus, A. R. Lebeck, M. Talluri, and D. A. Wood. Wisconsin architectural research tool set. *Computer Architecture News*, 21(4):8–10, August 1993.

[30] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Trans. Comput.*, C-38(12):1612–1630, Dec. 1989.

[31] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 1979.

[32] M. T. Kandemir, A. N. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A linear algebra framework for automatic determination of optimal data layouts. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):115–135, Feb. 1999.

[33] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. *The Omega Calculator and Library, version 1.1.0*, Nov. 1996.

[34] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. *The Omega Library Version 1.1.0 Interface Guide*, Nov. 1996.

[35] W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-3193, Department of Compute Science, University of Maryland, College Park, MD, Apr. 1993.

[36] W. Kelly and W. Pugh. Finding legal reordering transformations using mappings. Technical Report CS-TR-3297, Department of Compute Science, University of Maryland, College Park, MD, June 1994.

[37] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-index caches. *ACM Trans. Comput. Syst.*, 10(4):338–359, 1992.

[38] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Apr. 1991.

[39] A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, 27(10):15–26, Oct. 1994.

[40] A. R. Lebeck and D. A. Wood. Active memory: A new abstraction for memory system simulation. *ACM Transactions on Modeling and Computer Simulation*, 7(1):42–77, Jan. 1997.

[41] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languagess*, pages 201–214, Paris, France, Jan. 1997.

[42] V. Loechner. *PolyLib: A Library for Manipulating Parameterized Polyhedra*, Mar. 1999.

[43] M. Martonosi, A. Gupta, and T. Anderson. Memspy: Analyzing memory system bottlenecks in programs. In *SIGMETRICS92*, pages 1–12, June 1992.

[44] K. S. McKinley and O. Temam. Quantifying loop nest locality using SPEC'95 and the Perfect benchmarks. *ACM Trans. Comput. Syst.*, 17(4):288–336, Nov. 1999.

[45] N. Mitchell, L. Carter, J. Ferrante, and K. Högstedt. Quantifying the multi-level nature of tiling interactions. In *Languages and Compilers for Parallel Computing: 10th Annual Workshop, LCPC'97*, number 1366 in Lecture Notes in Computer Science, pages 1–15. Springer, 1998.

[46] D. C. Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. *J. Comput. Syst. Sci.*, 16(3):323–332, July 1978.

[47] Y. Paek, J. Hoeflinger, and D. Padua. Simplification of array access patterns for compiler optimizations. In *Proceedings of ACM PLDI*, volume 33, pages 60–71, May 1998.

[48] A. K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, Houston, TX, May 1989. Available as technical report CRPC-TR89009.

[49] W. Pugh. Counting solutions to Presburger formulas: How and why. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 121–134, Orlando, FL, June 1994.

[50] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 38–49, Montreal, Canada, June 1998.

[51] G. Rivera and C.-W. Tseng. Eliminating conflict misses for high performance architectures. In *Proceedings of the 1998 International Conference on Supercomputing*, pages 353–360, Melbourne, Australia, July 1998.

[52] U. Schöning. Complexity of Presburger arithmetic with fixed quantifier dimension. *Theory of Computing Systems*, 30:423–428, 1997.

[53] N. Shibata, K. Okana, T. Higashino, and K. Taniguchi. A decision algorithm dor prenex form rational Presburger sentences based on combinatorial geometry. In *Proceedings of the 2nd International Conference on Discrete Mathematics and Theoretical Computer Science and the 5th Australasian Theory Symposium (DMTCS'99+CATS'99)*, pages 344–359, Jan. 1999.

[54] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[55] The Stanford Compiler Group. *SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers*. http://suif.stanford.edu.

[56] R. A. Sugumar and S. G. Abraham. Efficient simulation of multiple cache configurations using binomial trees. *Technical Report CSE-TR-111-91*, 1991.

[57] D. Thiebaut and H. Stone. Footprints in the cache. *ACM Trans. Comput. Syst.*, 5(4):305–329, Nov. 1987.

[58] D. A. B. Weikle, S. A. McKee, and W. A. Wulf. Caches as filters: A new approach to cache analysis. In *MASCOTS'98, Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, July 1998.

[59] D. A. B. Weikle, K. Skadron, S. A. McKee, and W. A. Wulf. Caches as filters: A unifying model for memory hierarchy analysis. Technical Report CS-2000-16, University of Virginia, June 2000.

[60] V. Weispfenning. Complexity and uniformity of elimination in Presburger arithmetic. In *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation*, pages 48–53, Kihei, Maui, HI, July 1997.

[61] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Canada, June 1991.

[62] M. J. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing'89*, pages 655–664, Reno, NV, Nov. 1989.

[63] D. A. Wood, M. D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. In *Proceedings of ACM SIGMETRICS*, May 1991.

[64] H. Zhang and M. Martonosi. Mathematical cache miss analysis for pointer data structures. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*, Portsmouth, VA, Mar. 2001. CD-ROM.